Write a java program to implement the queue data structure.

A Queue is a linear data structure that follows the FIFO (First In, First Out) principle. This means the element that is inserted first is the one to be removed first — like people standing in a line (the one who comes first gets served first).

Objectives

1. Develop problem-solving skills
2. Practice core Java concepts
3. Learn how to handle overflow and underflow
4. Prepare for technical interviews and exams

Sources code:

```
import java.util.Scanner;

public class ArrayQueue {

   int front = -1, rear = -1, size;

   int[] q;


   ArrayQueue(int s) {

      size = s;

      q = new int[size];

   }


   void enqueue(int val) {

      if (rear == size - 1)

         System.out.println("Queue Full");

      else {

         if (front == -1) front = 0;

         q[++rear] = val;

         System.out.println(val + " enqueued");
```

```
    }
}

void dequeue() {
    if (front == -1 || front > rear)
        System.out.println("Queue Empty");
    else
        System.out.println(q[front++] + " dequeued");
}

void display() {
    if (front == -1 || front > rear)
        System.out.println("Queue Empty");
    else {
        System.out.print("Queue: ");
        for (int i = front; i <= rear; i++)
            System.out.print(q[i] + " ");
        System.out.println();
    }
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    ArrayQueue q = new ArrayQueue(3);
    q.enqueue(10);
    q.enqueue(20);
    q.display();
```

```
        q.dequeue();
        q.display();
        sc.close();
    }
}
```

Outputs:

```
10 enqueued
20 enqueued
Queue: 10 20
10 dequeued
Queue: 20
```

Write a java program for quick sort.

Quick Sort is a Divide and Conquer algorithm. It works by selecting a pivot element, and then partitioning the array such that:

- All elements less than or equal to the pivot go to the left of it.
- All elements greater than the pivot go to the right of it.

Objectives of the Quick Sort Program

1. Understand Divide and Conquer
2. Implement Recursion Practically
3. Learn Partitioning Techniques
4. Optimize Sorting Performance
5. Boost Analytical Thinking

Sources Code:

```
import java.util.Scanner;


public class QuickSort {
   static void quickSort(int[] a, int low, int high) {
      if (low < high) {
         int pi = partition(a, low, high);
         quickSort(a, low, pi - 1);
         quickSort(a, pi + 1, high);
      }
   }


   static int partition(int[] a, int low, int high) {
      int pivot = a[high], i = low - 1;
      for (int j = low; j < high; j++) {
         if (a[j] <= pivot) {
            int t = a[++i]; a[i] = a[j]; a[j] = t;
```

```
        }
    }
    int t = a[i + 1]; a[i + 1] = a[high]; a[high] = t;
    return i + 1;
}

static void print(int[] a) {
    for (int val : a) System.out.print(val + " ");
    System.out.println();
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter size: ");
    int n = sc.nextInt(), a[] = new int[n];
    System.out.println("Enter elements:");
    for (int i = 0; i < n; i++) a[i] = sc.nextInt();

    System.out.print("Original: ");
    print(a);

    quickSort(a, 0, n - 1);

    System.out.print("Sorted: ");
    print(a);
    sc.close();
}
```

}

Outputs:

```
Enter size: 8
Enter elements:
22 11 88 66 55 77 33 44
Original: 22 11 88 66 55 77 33 44
Sorted: 11 22 33 44 55 66 77 88
```

Write a java program to implement the stack data structure.

A Stack is a linear data structure that serves as a collection of elements, with two main operations:

- Push: Adds an element to the top of the stack.
- Pop: Removes the element from the top of the stack.

Objectives

- Understand how a stack operates using push and pop operations.
- Implement stack using a Java class and array.
- Learn how to handle overflow and underflow conditions.

Sources Code:

```
class Stack {

   int maxSize = 5;

   int[] stack = new int[maxSize];

   int top = -1;


   void push(int val) {

      if (top == maxSize - 1)

         System.out.println("Stack Overflow");

      else

         stack[++top] = val;

   }


   void pop() {

      if (top == -1)

         System.out.println("Stack Underflow");

      else

         System.out.println("Popped: " + stack[top--]);

   }
```

```
    void display() {
        if (top == -1)
            System.out.println("Stack is empty");
        else {
            System.out.print("Stack: ");
            for (int i = 0; i <= top; i++)
                System.out.print(stack[i] + " ");
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Stack s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        s.display();
        s.pop();
        s.display();
    }
}
```

Outputs: Stack: 10 20 30

Popped: 30

Stack: 10 20

Write an algorithm to find the factorial of n number using recursion.

Algorithm: Recursive Factorial

Step 1: Start
Step 2: Define a function `factorial(n)`
If `n == 0` or `n == 1`, return 1
Else, return `n * factorial(n - 1)`
Step 3: Read input number `n`
Step 4: Call `factorial(n)` and store the result
Step 5: Print the result
Step 6: End

```
import java.util.Scanner;

public class FactorialRecursion {

    static int factorial(int n) {

        if (n == 0 || n == 1)

            return 1;          // Base case

        return n * factorial(n - 1);  // Recursive case

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter a number: ");

        int num = sc.nextInt();

        System.out.println("Factorial of " + num + " is " + factorial(num));

        sc.close();

    }

}
```

Outputs: Enter a number: 5

Factorial of 5 is 120

Write a java program to implement the binary tree.

A Binary Tree is a tree data structure where each node has at most two children (left and right). Traversal like inorder helps to visit nodes in a sorted manner (for BSTs).

Sources code:

```
class Node {
   int data;
   Node left, right;

   Node(int value) {
      data = value;
      left = right = null;
   }
}

public class BinaryTree {
   Node root;
   void inorder(Node node) {
      if (node == null) return;
      inorder(node.left);
      System.out.print(node.data + " ");
      inorder(node.right);
   }

   public static void main(String[] args) {
      BinaryTree tree = new BinaryTree();

      // Manually create nodes and link
```

```
        tree.root = new Node(10);
        tree.root.left = new Node(5);
        tree.root.right = new Node(15);
        tree.root.left.left = new Node(3);
        tree.root.left.right = new Node(7);

        System.out.print("Inorder traversal: ");
        tree.inorder(tree.root);
    }
}
```

Output: 3 5 7 10 15

Write a java program to implement the linked lists operation.

A Linked List is a fundamental linear data structure where elements (called nodes) are stored in a sequence, but unlike arrays, the nodes are not stored at contiguous memory locations.

Objectives

1. Understand the concept of dynamic data structures and how they differ from arrays.
2. Create a node structure and manage node references (pointers) effectively.
3. Learn memory management concepts in the context of linked lists.
4. Prepare for advanced data structures like stacks, queues, and trees, which often use linked list concepts.

Sources code:

```
class Node {
   int data;
   Node next;

   Node(int val) {
      data = val;
      next = null;
   }
}

public class LinkedList {
   Node head;

   void insert(int val) {
      Node newNode = new Node(val);
      if (head == null) {
         head = newNode;
         return;
```

```
    }
    Node temp = head;
    while (temp.next != null)
        temp = temp.next;
    temp.next = newNode;
}
void delete(int val) {
    if (head == null) return;
    if (head.data == val) {
        head = head.next;
        return;
    }
    Node temp = head;
    while (temp.next != null && temp.next.data != val)
        temp = temp.next;
    if (temp.next != null)
        temp.next = temp.next.next;
}
void display() {
    Node temp = head;
    if (temp == null) {
        System.out.println("List is empty");
        return;
    }
    while (temp != null) {
        System.out.print(temp.data + " -> ");
        temp = temp.next;
```

```
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.display();

        list.delete(20);
        list.display();
    }
}
```

Outputs:

```
10 -> 20 -> 30 -> null
10 -> 30 -> null
```

Write a java program to implement the Breadth First Traversal.

Breadth First Traversal (BFS) is an important graph traversal algorithm used to explore nodes (vertices) and edges of a graph systematically.

Sources code:

```
import java.util.*;

public class BreadthFirstTraversal {

   private int vertices;

   private LinkedList<Integer>[] adjList;


   public BreadthFirstTraversal(int v) {

      vertices = v;

      adjList = new LinkedList[v];

      for (int i = 0; i < v; ++i)

         adjList[i] = new LinkedList<>();

   }

   public void addEdge(int u, int v) {

      adjList[u].add(v);

   }

   public void bfs(int start) {

      boolean[] visited = new boolean[vertices];

      Queue<Integer> queue = new LinkedList<>();


      visited[start] = true;

      queue.add(start);


      System.out.print("Breadth First Traversal starting from vertex " + start + ": ");
```

```
        while (!queue.isEmpty()) {
            int current = queue.poll();
            System.out.print(current + " ");

            for (int neighbor : adjList[current]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }
    }

    public static void main(String[] args) {
        BreadthFirstTraversal graph = new BreadthFirstTraversal(5);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.bfs(0);
    }
}
```

Outputs: Breadth First Traversal starting from vertex 0: 0 1 2 3 4

Write a java program to implement the Breadth First Traversal.

A Circular Queue is a linear data structure that follows the FIFO (First In First Out) principle, but unlike a regular queue, the last position is connected back to the first to make a circle.

Objectives:

- Enqueue (Insertion)
- Dequeue (Deletion)
- Displaying the queue.

Sources code:

```
import java.util.Scanner;

public class CircularQueue {

   int front, rear, size;

   int[] queue;


   public CircularQueue(int size) {

      this.size = size;

      queue = new int[size];

      front = rear = -1;

   }


   void enqueue(int value) {

      if ((rear + 1) % size == front) {

         System.out.println("Queue is Full!");

      } else {

         if (front == -1) front = 0;

         rear = (rear + 1) % size;

         queue[rear] = value;

         System.out.println(value + " inserted");
```

```
    }
}

void dequeue() {
    if (front == -1) {
        System.out.println("Queue is Empty!");
    } else {
        System.out.println(queue[front] + " deleted");
        if (front == rear) {
            front = rear = -1; // Queue becomes empty
        } else {
            front = (front + 1) % size;
        }
    }
}

void display() {
    if (front == -1) {
        System.out.println("Queue is Empty!");
        return;
    }
    System.out.print("Queue: ");
    int i = front;
    while (true) {
        System.out.print(queue[i] + " ");
        if (i == rear) break;
        i = (i + 1) % size;
```

```
    }
    System.out.println();
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    CircularQueue cq = new CircularQueue(5);

    while (true) {
        System.out.println("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit");
        System.out.print("Choose: ");
        int choice = sc.nextInt();

        switch (choice) {
            case 1:
                System.out.print("Enter value: ");
                int val = sc.nextInt();
                cq.enqueue(val);
                break;
            case 2:
                cq.dequeue();
                break;
            case 3:
                cq.display();
                break;
            case 4:
                System.out.println("Bye!");
```

```
                return;
            default:
                System.out.println("Invalid!");
        }
    }
  }
}
```

Outputs:

```
1 → Enqueue → 10
1 → Enqueue → 20
3 → Display
2 → Dequeue
3 → Display
```

Write a java program to implement the Depth First Traversal.

Depth First Traversal (DFS) is a graph traversal algorithm where you go as deep as possible before backtracking. It uses recursion or a stack.

Sources code:

```
import java.util.*;

public class DFSTraversal {
   private int V;  // Number of vertices
   private List<List<Integer>> adj; // Adjacency list


   public DFSTraversal(int v) {
      V = v;
      adj = new ArrayList<>();
      for (int i = 0; i < v; i++)
         adj.add(new ArrayList<>());
   }


   void addEdge(int v, int w) {
      adj.get(v).add(w);
   }


   void DFSUtil(int v, boolean[] visited) {
      visited[v] = true;
      System.out.print(v + " ");
      for (int neighbor : adj.get(v)) {
         if (!visited[neighbor])
            DFSUtil(neighbor, visited);
      }
```

```
    }

    void DFS(int start) {
        boolean[] visited = new boolean[V];
        System.out.print("DFS starting from node " + start + ": ");
        DFSUtil(start, visited);
    }

    public static void main(String[] args) {
        DFSTraversal graph = new DFSTraversal(5);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);

        graph.DFS(0);
    }
}
```

Outputs: DFS starting from node 0: 0 1 3 4 2

Write a java program to implement the minimal spanning tree (using Kruskal's Algorithm).

MST connects all nodes with minimum edge weight without forming a cycle.
Kruskal's Algorithm.

- Sort edges by weight
- Add the smallest edge that doesn't make a cycle (using union-find).

Sources code:

```
import java.util.*;

class Edge implements Comparable<Edge> {

   int u, v, w;

   Edge(int u, int v, int w) { this.u = u; this.v = v; this.w = w; }

   public int compareTo(Edge e) { return this.w - e.w; }

}

public class Kruskal {

   static int[] parent;


   static int find(int i) {

      if (parent[i] != i) parent[i] = find(parent[i]);

      return parent[i];

   }

   static void union(int a, int b) {

      parent[find(a)] = find(b);

   }

   public static void main(String[] args) {
```

```
        int V = 4;
        Edge[] edges = {
            new Edge(0, 1, 10), new Edge(0, 2, 6),
            new Edge(0, 3, 5), new Edge(1, 3, 15),
            new Edge(2, 3, 4)
        };
        Arrays.sort(edges);
        parent = new int[V];
        for (int i = 0; i < V; i++) parent[i] = i;

        System.out.println("MST Edges:");
        for (Edge e : edges) {
            if (find(e.u) != find(e.v)) {
                System.out.println(e.u + " - " + e.v + " : " + e.w);
                union(e.u, e.v);
            }
        }
    }
}
```

Outputs: MST Edges:

2 - 3 : 4

0 - 3 : 5

0 - 1 : 10